intel®

APPLICATION NOTE

AP-360

November 1992

3

28F008SA Software Drivers

BRIAN DIPERT
MCD MARKETING APPLICATIONS

Order Number: 292095-002

28F008SA Software Drivers

CONTENTS

CONTENTS

1.0 INTRODUCTION

This application note provides example software code for byte writing, block erasing and otherwise controlling Intel's 28F008SA 8 Mbit symmetrically blocked FlashFile™ Memory family. Two programming languages are provided; high-level "C" for multi-platform support, and ASM-86 assembly. In many cases, the driver routines can be inserted "as is" into the main body of code being developed by the system software engineer. The text accompanying each routine describes the existing code and suggests area for possible alteration to fit specific applications. These explanations, along with in-line commenting, minimize driver modification efforts.

Companion product datasheets for the 28F008SA and 28F008SA-L are valuable reference documents. Datasheets should be reviewed in conjunction with this application note for a complete understanding of the devices. AP-359, "28F008SA Hardware Interfacing" is the hardware-oriented application note equivalent for these devices and can also be referenced.

The internal automation of the 28F008SA makes software timing loops unnecessary and results in platform-independent code. This software is designed to be executed in any type of memory and with all processor clock rates. "C" code can be used with many microprocessors and microcontrollers, while ASM-86 assembly code provides the smallest code "kernal" for Intel microprocessors and embedded processors.

2.0 ASM-86 DRIVERS

```
;       Copyright Intel Corporation, 1992
;       Brian Dipert, Intel Corporation, February 8, 1992, Revision 1.0

;       Revision History: Rev 1.0

;       The following code controls byte write of data to a single 28F008SA (x8 write)
;       DS:[SI] points to the data to be written, ES:[DI] is the location to be written
;       In protected mode operation, DS and ES reference a descriptor
;       Register AX is modified by this procedure
WRITE_SETUP         EQU     40H
READ_ID             EQU     90H
INTEL_ID            EQU     89H
DEVICE_ID           EQU     0A2H
DEVICE_ID2          EQU     0A1H
READY               EQU     80H
W_ERR_FLAG          EQU     10H
VPP_FLAG            EQU     08H
;       Insert code here to ramp Vpp and disable component /PWD input. If a string of bytes is
;               to be written at one time, Vpp ramp to 12V and ID check need only occur once,
;               before the first byte is written
        MOV     AX,     "Address 0 for target 28F008SA-segment"
                                        ; Initialize pointer to 28F0008SA address 0
        MOV     ES,     AX
        MOV     DI,     "Address 0 for target 28F008SA-offset"
        MOV     BYTE PTR ES:[DI],   READ_ID     ; Write Inteligent Identifier command
        CMP     BYTE PTR ES:[DI],   INTEL_ID    ; Does manufacturer ID read correctly?
        JNZ     W_BYT_ID_ERR
        MOV     DI,     "Address 1 for target 28F008SA-offset"
                                        ; Initialize pointer to 28F008SA address 1
        CMP     BYTE PTR ES:[DI],   DEVICE_ID   ; Does device ID read correctly?
        JZ      W_BYT_ID_PASS
        CMP     BYTE PTR ES:[DI],   DEVICE_ID2
        JNZ     W_BYT_ID_ERR

W_BYT_ID_PASS:
        MOV     AX,     "Byte write destination address-segment"
                                        ; Initialize pointer to byte write dest. address
        MOV     ES,     AX
        MOV     DI,     "Byte write destination address-offset"
        MOV     BYTE PTR ES:[DI],   WRITE_SETUP ; Write byte write setup command
        MOV     AL,     DS:[SI]                 ; Load AL with data to write
        MOV     ES:[DI],  AL                    ; Write to device
W_BYT_LOOP:
        TEST    BYTE PTR ES:[DI],   READY       ; Read 28F008SA Status Register
        JZ      W_BYT_LOOP                      ; Loop until bit 7 = 1

        TEST    BYTE PTR ES:[DI],   (W_ERR_FLAG OR VPP_FLAG)
        JZ      W_BYT_CONT                      ; Success!

        TEST    BYTE PTR ES:[DI],   W_ERR_FLAG  ; Check Status Register bit 4
        JNZ     W_BYT_ERR                       ; Jump if = 1, Byte Write Error

        TEST    ES:[DI],  VPP_FLAG              ; Check Status Register bit 3
        JNZ     W_BYT_VPP                       ; Jump if = 1, Vpp Low Error

W_BYT_ID_ERR:
;               Insert code to service improper device ID read error here.
;               Is 28F008SA /PWD input disabled? Is Vcc applied to the 28F008SA?
W_BYT_ERR:
;               Insert code to service byte write error here
W_BYT_VPP:
;               Insert code to service byte write Vpp low error here
W_BYT_CONT:
;               Code continues from this point.....
```

This routine writes a byte of data to a single 28F008SA. Note the use of BYTE PTR notation to force x8 accesses. If a string of bytes is to be written at one time, the V_{PP} ramp up, $\overline{PWD}$ disable and device ID checks need only be done before the first byte write attempt. Additionally, when writing multiple bytes at once, examination of bits other than bit 7 (WSM Status) need only occur after the last byte write has completed. The Status Register retains any error bits until the Clear Status Register command is written.

```
;       The following code controls byte write of data to a pair of 28F008SAs (x16 write)
;       DS:[SI] points to the data to be written, ES:[DI] is the location to be written
;       In protected mode operation, DS and ES reference a descriptor
;       Register AX is modified by this procedure
WRITE_SETUP         EQU     40H
READ_ID             EQU     90H
INTEL_ID            EQU     89H
DEVICE_ID           EQU     0A2H
DEVICE_ID2          EQU     0A1H
READY               EQU     80H
W_ERR_FLAG          EQU     10H
VPP_FLAG            EQU     08H
;       Insert code here to ramp Vpp and disable component /PWD inputs. If a string of words is
;           to be written at one time, Vpp ramp to 12V and ID check need only occur once,
;           before the first word is written
        MOV     AX,     "Address 0 for target 28F008SA-segment"
                                    ; Initialize pointer to 28F008SA address 0
        MOV     ES,     AX
        MOV     DI,     "Address 0 for target 28F008SA-offset"
        MOV     ES:[DI], ((READ_ID SHL 8) OR READ_ID)
                                    ; Write Inteligent Identifier command
        CMP     ES:[DI], ((INTEL_ID SHL 8) OR INTEL_ID)
                                    ; Does manufacturer ID read correctly?
        JNZ     W_WRD_ID_ERR
        MOV     DI,     "Address 1 for target 28F008SA-offset"
                                    ; Initialize pointer to 28F008SA address 1
        CMP     ES:[DI], ((DEVICE_ID SHL 8) OR DEVICE_ID)
                                    ; Does device ID read correctly?
        JZ      W_WRD_ID_PASS
        CMP     ES:[DI], ((DEVICE_ID2 SHL 8) OR DEVICE_ID2)
        JNZ     W_WRD_ID_ERR

W_WRD_ID_PASS:
        MOV     AX,     "Byte write destination address-segment"
                                    ; Initialize pointer to byte write dest. address
        MOV     ES,     AX
        MOV     DI,     "Byte write destination address-offset"
        MOV     ES:[DI], ((WRITE_SETUP SHL 8) OR WRITE_SETUP)     ; Write byte write setup command
        MOV     AX,     DS:[SI]     ; Load AX with data to write
        MOV     ES:[DI], AX         ; Write to devices
W_WRD_LOOP:
        TEST    ES:[DI], ((READY SHL 8) OR READY)  ; Read 28F008SA Status Registers
        JZ      W_WRD_LOOP          ; Loop until bit 7 = 1

        TEST    ES:[DI], (((W_ERR_FLAG OR VPP_FLAG) SHL 8) OR (W_ERR_FLAG OR VPP_FLAG))
        JZ      W_WRD_CONT          ; Success!

        MOV     AX,     ES:[DI]     ; Load Status Register data into AX
        TEST    AL,     W_ERR_FLAG  ; Check Status Register bit 4 (low byte)
        JNZ     W_WRD_ERR           ; Jump if = 1
        TEST    AH,     W_ERR_FLAG  ; Check Status Register bit 4 (high byte)
        JNZ     W_WRD_ERR           ; Jump if = 1

        TEST    AL,     VPP_FLAG    ; Check Status Register bit 3 (low byte)
        JNZ     W_WRD_VPP           ; Jump if = 1
        TEST    AH,     VPP_FLAG    ; Check Status Register bit 3 (high byte)
        JNZ     W_WRD_VPP           ; Jump if = 1

W_WRD_ID_ERR:
;       Insert code to service improper device ID read error here.
;       Are 28F008SA /PWD inputs disabled? Is Vcc applied to the 28F008SAs?

W_WRD_ERR:
;       Insert code to service byte write error here
W_WRD_VPP:
;       Insert code to service byte write Vpp low error here
W_WRD_CONT:
;       Code continues from this point.....
```

This routine writes a word of data to a pair of 28F008SAs. Note that all constants have been "OR'd" for parallel read/write of two devices at once. If a string of words is to be written at one time, the V_{PP} ramp up, $\overline{PWD}$ disable and device ID checks need only be done before the first word write attempt. Additionally, when writing multiple words at once, examination of bits other than bit 7 (WSM Status) need only occur after the last word write has completed. The Status Register retains any error bits until the Clear Status Register command is written.

```
;         The following code controls block erase of a single 28F008SA (x8 block erase)
;         ES:[DI] points to the block to be erased
;         In protected mode operation, ES references a descriptor
;         Register AX is modified by this procedure
ERASE_SETUP         EQU       20H
ERASE_CONFIRM       EQU       0D0H
READ_ID             EQU       90H
INTEL_ID            EQU       89H
DEVICE_ID           EQU       0A2H
DEVICE_ID2          EQU       0A1H
READY               EQU       80H
E_ERR_FLAG          EQU       20H
E_CMD_FLAG          EQU       30H
VPP_FLAG            EQU       08H
;         Insert code here to ramp Vpp and disable component /PWD input. If a string of blocks is
;                   to be erased at one time, Vpp ramp to 12V and ID check need only occur once,
;                   before the first block is erased
          MOV       AX,       "Address 0 for target 28F008SA-segment"
                                          ; Initialize pointer to 28F008SA address 0
          MOV       ES,       AX
          MOV       DI,       "Address 0 for target 28F008SA-offset"
          MOV       BYTE PTR ES:[DI],   READ_ID     ; Write Inteligent Identifier command
          CMP       BYTE PTR ES:[DI],   INTEL_ID    ; Does manufacturer ID read correctly?
          JNZ       E_BYT_ID_ERR
          MOV       DI,       "Address 1 for target 28F008SA-offset"
                                          ; Initialize pointer to 28F008SA address 1
          CMP       BYTE PTR ES:[DI],   DEVICE_ID   ; Does device ID read correctly?
          JZ        E_BYT_ID_PASS
          CMP       BYTE PTR ES:[DI],   DEVICE_ID2
          JNZ       E_BYT_ID_ERR

E_BYT_ID_PASS:
          MOV       AX,       "Block erase destination address-segment"
                                          ; Initialize pointer to block erase dest.address
          MOV       ES,       AX
          MOV       DI,       "Block erase destination address-offset"
          MOV       BYTE PTR ES:[DI],   ERASE_SETUP     ; Write block erase setup command
          MOV       BYTE PTR ES:[DI],   ERASE_CONFIRM   ; Write block erase confirm command
E_BYT_LOOP:
          TEST      BYTE PTR ES:[DI],   READY       ; Read 28F0008SA Status Register
          JZ        E_BYT_LOOP                    ; Loop until bit 7 = 1

          TEST      BYTE PTR ES:[DI],   (E_CMD_FLAG OR VPP_FLAG)
          JZ        E_BYT_CONT                    ; Success!

          TEST      BYTE PTR ES:[DI],   E_CMD_FLAG ; Check Status Register bits 4 and 5
          JNZ       E_BYT_CMD_ERR                 ; Jump if = 1

          TEST      BYTE PTR ES:[DI],   E_ERR_FLAG ; Check Status Register bit 5
          JNZ       E_BYT_ERR                     ; Jump if = 1

          TEST      BYTE PTR ES:[DI],   VPP_FLAG   ; Check Status Register bit 3
          JNZ       E_BYT_VPP                     ; Jump if = 1

E_BYT_ID_ERR:
;         Insert code to service improper device ID read error here.
;         Is 28F008SA /PWD input disabled? Is Vcc applied to the 28F008SA?
E_BYT_CMD_ERR:
;         Insert code to service block erase command sequence error here
;         (setup followed by a command other than confirm)
E_BYT_ERR:
;         Insert code to service block erase error here
E_BYT_VPP:
;         Insert code to service block erase Vpp low error here
E_BYT_CONT:
;         Code continues from this point.....
```

This routine erases a block of a single 28F00SA. Note the use of BYTE PTR notation to force x8 accesses. If a string of blocks is to be erased at one time, the V_{PP} ramp up, $\overline{PWD}$ disable and device ID checks need only be down before the first block erase attempt. Additionally, when erasing multiple blocks at once, examination of bits other than bit 7 (WSM Status) need only occur after the last block erase has completed. The Status Register retains any error bits until the Clear Status Register command is written.

```
;       The following code controls block erase of a pair of 28F008SAs (x16 block erase)
;       ES:[DI] points to the blocks to be erased
;       In protected mode operation, ES references a descriptor
;       Register AX is modified by this procedure
ERASE_SETUP         EQU     20H
ERASE_CONFIRM       EQU     0D0H
READ_ID             EQU     90H
INTEL_ID            EQU     89H
DEVICE_ID           EQU     0A2H
DEVICE_ID2          EQU     0A1H
READY               EQU     80H
E_ERR_FLAG          EQU     20H
E_CMD_FLAG          EQU     30H
VPP_FLAG            EQU     08H
;       Insert code here to ramp Vpp and disable component /PWD inputs. If a string of blocks is
;               to be erased at one time, Vpp ramp to 12V and ID check need only occur once,
;               before the first block pair is erased
        MOV     AX,     "Address 0 for target 28F008SA-segment"
                                        ; Initialize pointer to 28F008SA address 0
        MOV     ES,     AX
        MOV     DI,     "Address 0 for target 28F008SA-offset"
        MOV     ES:[DI], ((READ_ID SHL 8) OR READ_ID)
                                        ; Write Inteligent Identifier command
        CMP     ES:[DI], ((INTEL_ID SHL 8) OR INTEL_ID)
                                        ; Does manufacturer ID read correctly?
        JNZ     E_WRD_ID_ERR
        MOV     DI,     "Address 1 for target 28F008SA-offset"
                                        ; Initialize pointer to 28F008SA address 1
        CMP     ES:[DI], ((DEVICE_ID SHL 8) OR DEVICE_ID)
                                        ; Does device ID read correctly?
        JZ      E_WRD_ID_PASS
        CMP     ES:[DI], ((DEVICE_ID2 SHL 8) OR DEVICE_ID2)
        JNZ     E_WRD_ID_ERR

E_WRD_ID_PASS:
        MOV     AX,     "Block erase destination address-segment"
                                        ; Initialize pointer to block erase dest. address
        MOV     ES,     AX
        MOV     DI,     "Block erase destination address-offset"
        MOV     ES:[DI], ((ERASE_SETUP SHL 8) OR ERASE_SETUP)
                                        ; Write block erase setup command
        MOV     ES:[DI], ((ERASE_CONFIRM SHL 8) OR ERASE_CONFIRM)
                                        ; Write block erase confirm command
E_WRD_LOOP:
        TEST    ES:[DI], ((READY SHL 8) OR READY)  ; Read 28F008SA Status Registers
        JZ      E_WRD_LOOP              ; Loop until bit 7 = 1

        TEST    ES:[DI], (((E_CMD_FLAG OR VPP_FLAG) SHL 8) OR (E_CMD_FLAG OR VPP_FLAG))
        JZ      E_WRD_CONT              ; Success!

        MOV     AX,     ES:[DI]         ; Load Status Register data into AX
        TEST    AL,     E_CMD_FLAG      ; Check Status Reg bits 4 and 5 (low byte)
        JNZ     E_WRD_CMD_ERR           ; Jump if = 1
        TEST    AH,     E_CMD_FLAG      ; Check Status Register bits 4 and 5 (high byte)
        JNZ     E_WRD_CMD_ERR           ; Jump if = 1

        TEST    AL,     E_ERR_FLAG      ; Check Status Register bit 5 (low byte)
        JNZ     E_WRD_ERR               ; Jump if = 1
        TEST    AH,     E_ERR_FLAG      ; Check Status Register bit 5 (high byte)
        JNZ     E_WRD_ERR               ; Jump if = 1

        TEST    AL,     VPP_FLAG        ; Check Status Register bit 3 (low byte)
        JNZ     E_WRD_VPP               ; Jump if = 1
        TEST    AH,     VPP_FLAG        ; Check Status Register bit 3 (high byte)
        JNZ     E_WRD_VPP               ; Jump if = 1

E_WRD_ID_ERR:
;       Insert code to service improper device ID read error here.
;       Are 28F008SA /PWD inputs disabled? Is Vcc applied to the 28F008SAs?
E_WRD_CMD_ERR:
;       Insert code to service block erase command sequence error here
;       (setup followed by a command other than confirm)
E_WRD_ERR:
;       Insert code to service block erase error here
E_WRD_VPP:
;       Insert code to service block erase Vpp low error here
E_WRD_CONT:
;       Code continues from this point.....
```

This routine erases a block pair of two 28F008SAs. Note that all constants have been "OR'd" for parallel read/write of two devices at once. If a string of block pairs is to be erased at one time, the V_{PP} ramp up, $\overline{PWD}$ disable and device ID checks need only be done before the first block pair erase attempt. Additionally, when erasing multiple block pairs at once, examination of bits other than bit 7 (WSM Status) need only occur after the last block pair erase has completed. The Status Register retains any error bits until the Clear Status Register command is written.

3.0 'C' DRIVERS

```
/*****************************************************************************************************/
/*      Copyright Intel Corporation, 1992                                                            */
/*      Brian Dipert, Intel Corporation, May 7, 1992, Revision 2.1                                   */
/*      The following drivers control the Command and Status Registers of the 28F008SA Flash         */
/*            Memory to drive byte write, block erase, Status Register read and clear and            */
/*            array read algorithms. Sample Vpp and /PWD control blocks are also included,           */
/*            as are example programs combining drivers into full algorithms                         */
/*      The functions listed below are included:                                                     */
/*            erasbgn(): Begins block erasure                                                        */
/*            erassusp(): Suspends block erase to allow reading data from a block of the             */
/*                  28F008SA other than that being erased                                            */
/*            erasres(): Resumes block erase if suspended                                            */
/*            end(): Polls the Write State Machine to determine if block erase or byte write         */
/*                  have completed                                                                   */
/*            eraschk(): Executes full status check after block erase completion                     */
/*            writebgn(): Begins byte write                                                          */
/*            writechk(): Executes full status check after byte write completion                     */
/*            idread(): Reads and returns the manufacturer and device IDs of the target              */
/*                  28F008SA                                                                         */
/*            statrd(): Reads and returns the contents of the Status Register                        */
/*            statclr(): Clears the Status Register                                                  */
/*            rdmode (): Puts the 28F008SA in Read Array mode                                        */
/*            rdbyte (): Reads and returns a specified byte from the target 28F008SA                 */
/*            vppup(): Enables high voltage Vpph                                                     */
/*            vppdown(): Disables Vpph                                                               */
/*            pwden(): Enables active low signal /PWD                                                */
/*            pwddis(): Disables active low signal /PWD                                              */
/*                                                                                                   */
/*      Addresses are transferred to functions as pointers to far bytes (ie long integers). An       */
/*            alternate approach is to create a global array the size of the 28F008SA and            */
/*            located "over" the 28F008SA in the system memory map. Accessing specific               */
/*            locations of the 28F008SA is then accomplished by passing the chosen function          */
/*            an offset from the array base versus a specific address. Different                     */
/*            microprocessor architectures will require different array definitions; ie for          */
/*            the x86 architecture, define it as "byte eightmeg[16][10000]" and pass each            */
/*            function TWO offsets to access a specific location. MCS-96 architectures are          */
/*            limited to "byte eightmeg[10000]"; alternate approaches such as using port pins        */
/*            for paging will be required to access the full flash array                             */
/*                                                                                                   */
/*      To create a far pointer, a function such as MK_FP() can be used, given a segment and         */
/*            offset in the x86 architecture. I use Turbo-C; see your compiler reference             */
/*            manual for additional information.                                                     */
/*****************************************************************************************************/

/*****************************************************************************************************/
/*      Revision History: Rev 2.1                                                                    */
/*                                                                                                   */
/*      Changes From Revision 1.0 to Revision 2.0:                                                   */
/*            Added alternate 28F008SA device ID to routine idread()                                 */
/*                                                                                                   */
/*      Changes from 2.0 to 2.1: Revised the Erase Suspend algorithm to remove potential             */
/*            "infinite loop" caused by the WSM going "ready" after the system reads the             */
/*            Status Register, and before the system issues the Erase Suspend command                */
/*****************************************************************************************************/

typedef  unsigned char byte;
```

```
/*****************************************************************************************************/
/*      Function: Main                                                                               */
/*      Description: The following code shows examples of byte write and block                       */
/*                erase algorithms that can be modified to fit the specific application and          */
/*                hardware design                                                                    */
/*****************************************************************************************************/
main()
{
        byte far *address;
        byte data,status;

/*      The following code gives an example of a possible byte write algorithm.                      */
/*      Note that Vpp does not need to be cycled between byte writes when a string of byte           */
/*      writes occurs. Ramp Vpp to 12V before the first byte write and leave at 12V until after      */
/*      completion of the last byte write. Doing so minimizes Vpp ramp up-down delay and             */
/*      maximizes byte write throughput                                                              */
        vppup();
/*      "INSERT SOFTWARE DELAY FOR VPP RAMP IF REQUIRED"                                             */
        pwddis();
        address   = 0XxxxxxL;
        data      = 0Xyy;
        if (writebgn(data,address) == 1)
/*              "RECOVERY CODE-POWER NOT APPLIED (ID CHECK FAIL)"                                    */
        else
        {
                while (end(&status) )
                switch (writechk(status))
                {
                        case 0:
                                break;
                        case 1:
/*                              "RECOVERY CODE-VPP LOW DETECT ERROR"                                 */
                                break;
                        case 2:
/*                              "RECOVERY CODE-BYTE WRITE ERROR"                                     */
                                break;
                }
                statclr();
        }
        vppdown();
```

This "C" routine gives an example of combining lower-level functions (found in following pages) to complete a byte write. Routines vppup( ) and pwddis( ) enable the 28F008SA for byte write. Function writebgn( ) issues a byte write sequence to the device, end( ) detects byte write completion via Status Register bit 7, and writechk( ) analyzes Status Register bits 3–6 to determine byte write success. If a string of bytes is to be written at one time, V_{PP} ramp up and $\overline{PWD}$ disable need only be done before the first byte write attempt. Additionally, when writing multiple bytes at once, examination of bits other than bit 7 (WSM Ready) need only occur after the last byte write has completed. The Status Register retains any error bits until the Clear Status Register command is written.

```
/*      The following code gives an example of a possible block erase algorithm.                    */
/*      Note that Vpp does not need to be cycled between block erases when a string of block        */
/*      erases occurs. Ramp Vpp to 12V before the first block erase and leave at 12V until after   */
/*      completion of the last block erase. Doing so minimizes Vpp ramp up-down delay and           */
/*      maximizes block erase throughput                                                            */
        vppup();
/*      "INSERT SOFTWARE DELAY FOR VPP RAMP IF REQUIRED"                                            */
        pwddis();
        address   = 0XxxxxxL;
        if (erasbgn(address) == 1)
/*              "RECOVERY CODE-POWER NOT APPLIED (ID CHECK FAIL)"                                   */
        else
        {
                while (end(&status) )
                        ;
                switch (eraschk(status))
                {
                        case 0:
                                break;
                        case 1:
/*                              "RECOVERY CODE-VPP LOW DETECT ERROR"                                */
                                break;
                        case 2:
/*                              "RECOVERY CODE-BLOCK ERASE ERROR"                                   */
                                break;
                        case 3:
/*                              "RECOVERY CODE-ERASE SEQUENCE ERROR"                                */
                                break;
                }
                statclr();
        }
        vppdown();
}
```

This "C" routine gives an example of combining lower-level functions (found in following pages) to complete a block erase. Routines vppup( ) and pwddis( ) enable the 28F008SA for block erase. Function erasbgn( ) issues a block erase sequence to the device, end( ) detects block erase completion via Status Register bit 7, and eraschk( ) analyzes Status Register bits 3–6 to determine block erase success. If a string of blocks is to be erased at one time, V_{PP} ramp up and $\overline{PWD}$ disable need only be done before the first block erase attempt. Additionally, when erasing multiple blocks at once, examination of bits other than bit 7 (WSM Ready) need only occur after the last block erase has completed. The Status Register retains any error bits until the Clear Status Register command is written.

```
/*****************************************************************************************************/
/*        Function: Erasgbn                                                                          */
/*        Description: Begins erase of a block.                                                      */
/*        Inputs:   blckaddr: System address within the block to be erased                           */
/*        Outputs:  None                                                                             */
/*        Returns:  0 = Block erase successfully initiated                                           */
/*                  1 = Block erase not initiated (ID check error)                                   */
/*        Device Read Mode on Return: Status Register (ID if returns 1)                              */
/*****************************************************************************************************/

#define   ERASETUP  0X20              /* Erase Setup command                                          */
#define   ERASCONF  0XD0               /* Erase Confirm command                                       */

int erasbgn(blckaddr)

byte far *blckaddr;                    /* blckaddr is an address within the block to be erased       */

{
          byte mfgrid,deviceid;

          if (idread(&mfgrid,&deviceid)==1)  /* ID read error; device not powered up?                 */
                    return (1);
          *blckaddr = ERASETUP;          /* Write Erase Setup command to block address               */
          *blckaddr = ERASCONF;          /* Write Erase Confirm command to block address             */
          return (0);
}
```

Routine erasbgn( ) issues a block erase command sequence to a 28F008SA. It is passed the desired system address for the block to be erased. After calling idread( ), it writes the erase command sequence at the specified address. It returns "0" if block erase initiation was successful, and "1" if the ID read fails (device not powered up or $\overline{PWD}$ not disabled).

```
/*****************************************************************************************************/
/*        Function: Erassusp                                                                         */
/*        Description: Suspends block erase to read from another block                               */
/*        Inputs:   None                                                                             */
/*        Outputs:  None                                                                             */
/*        Returns:  0 = Block erase suspended                                                        */
/*                  1 = Error; Write State Machine not busy (block erase suspend not possible)       */
/*        Device Read Mode on Return: Status Register                                                */
/*****************************************************************************************************/

#define   RDYMASK   0X80              /* Mask to isolate the WSM Status bit of the Status Register */
#define   WSMRDY    0X80              /* Status Register value after masking, signifying that      */
                                      /*          the WSM is no longer busy                        */
#define   SUSPMASK  0X40              /* Mask to isolate the erase suspend status bit of the       */
                                      /*          Status Register                                  */
#define   ESUSPYES  0X40              /* Status Register value after masking, signifying that      */
                                      /*          block erase has been suspended                   */
#define   STATREAD  0X70              /* Read Status Register command                              */
#define   SYSADDR   0                 /* This constant can be initialized to any address within    */
                                      /*          the memory map of the target 28F008SA and is     */
                                      /*          alterable depending on the system architecture   */
#define   SUSPCMD   0XB0              /* Erase Suspend command                                     */

int erassusp()

{
          byte far *stataddr;         /* Pointer variable used to write commands to device         */

          stataddr  = (byte far *)SYSADDR;
          *stataddr = SUSPCMD;        /* Write Erase Suspend command to the device                 */
          *stataddr = STATREAD;       /* Write Read Status Register command..necessary in case     */
                                      /*          erase is already completed                       */
          while ((*stataddr & RDYMASK) != WSMRDY)
                    ;                 /* Will remain in while loop until bit 7 of the Status       */
                                      /*          Register goes to 1, signifying that              */
                                      /*          the WSM is no longer busy                        */
          if ((*stataddr & SUSPMASK) = ESUSPYES)
                    return (0);       /* Erase is suspended. . return code "0"                     */
          return (1);                 /*Erase has already completed; suspend not possible.         */
                                      /*          Error code "1"                                   */
}
```

Routine erassusp( ) issues the erase suspend command to a 28F008SA. It first makes sure the WSM is truly busy, then issues the erase suspend command and polls Status Register bits 7 and 6 until they indicate erase suspension. It returns "0" if block erase was successful, and "1" if the WSM was not busy when suspend was attempted.

```
/*****************************************************************************************/
/*      Function: Erasres                                                                */
/*      Description: Resumes block erase previously suspended                            */
/*      Inputs:   None                                                                   */
/*      Outputs:  None                                                                   */
/*      Returns:  0 = Block erase resumed                                                */
/*                1 = Error; Block erase not suspended when function called              */
/*      Device Read Mode on Return: Status Register                                      */
/*****************************************************************************************/

#define   RDYMASK   0X80              /* Mask to isolate the WSM Status bit of the Status Register */
#define   WSMRDY    0X80              /* Status Register value after masking, signifying that the  */
                                      /*       WSM is no longer busy                               */
#define   SUSPMASK  0X40              /* Mask to isolate the erase suspend status bit of the       */
                                      /*       Status Register                                     */
#define   ESUSPYES  0X40              /* Status Register value after masking, signifying that      */
                                      /*       block erase has been suspended                      */
#define   STATREAD  0X70              /* Read Status Register command                              */
#define   SYSADDR   0                 /* This constant can be initialized to any address within    */
                                      /*       the memory map of the target 28F008SA and is        */
                                      /*       alterable depending on the system architecture      */
#define   RESUMCMD  0XD0              /* Erase Resume command                                      */

int erasres()

{
        byte far *stataddr;           /* Pointer variable used to write commands to device         */

        stataddr  = (byte far *)SYSADDR;
        *stataddr = STATREAD;         /* Write Read Status Register command to 28F008SA            */
        if ((*stataddr & SUSPMASK) != ESUSPYES)
                return (1);           /* Block erase not suspended. Error code "1"                 */
        *stataddr = RESUMCMD;         /* Write Erase Resume command to the device                  */
        while ((*stataddr & SUSPMASK) == ESUSPYES)
                ;                     /* Will remain in while loop until bit 6 of the Status       */
                                      /*       Register goes to 0, signifying block                */
                                      /*       erase resumption                                    */
        while ((*stataddr & RDYMASK) == WSMRDY)
                ;                     /* Will remain in while loop until bit 7 of the Status       */
                                      /*       Register goes to 0, signifying that the WSM is      */
                                      /*       once again busy                                     */
        return (0);
}
```

Routine erasres( ) issues the erase resume command to a 28F008SA. It first makes sure the WSM is truly suspended, then issues the erase resume command and polls Status Register bits 7 and 6 until the indicate WSM resumption. It returns "0" if block erase resume was successful, and "1" if the WSM was not suspended when resumption was attempted.

```
/*****************************************************************************************************/
/*          Function: End                                                                            */
/*          Description: Checks to see if the WSM is busy (is byte write/block erase completed?)     */
/*          Inputs:         None                                                                     */
/*          Outputs:        statdata: Status Register data read from device                          */
/*          Returns:        0 = Byte Write/Block Erase completed                                     */
/*                          1 = Byte Write/Block Erase still in progress                             */
/*          Device Read Mode on Return: Status Register                                              */
/*****************************************************************************************************/
#define     RDYMASK         0X80                /* Mask to isolate the WSM Status bit of the Status    */
#define     WSMRDY          0X80                /* Register value after masking, signifying that the   */
                                                /*         WSM is no longer busy                       */
#define     STATREAD        0X70                /* Read Status Register command                        */
#define     SYSADDR         0                   /* This constant can be initialized to any address within*/
                                                /*         the memory map of the target 28F008SA and is*/
                                                /*         alterable depending on the system architecture */

int end(statdata)

byte *statdata;                                 /* Allows Status Register data to be passed back to the  */
                                                /*         main program for further analysis           */

{
          byte far *stataddr;                   /* Pointer variable used to write commands to device     */

          stataddr      = (byte far *)SYSADDR;
          *stataddr     = STATREAD;             /* Write Read Status Register command to 28F008SA        */
          if (((*statdata = *stataddr) & RDYMASK) != WSMRDY)
                        return (1);             /* Byte write/block erasure still in progress...code "1" */
          return (0);                           /* Byte write/block erase attempt completed...code "0"   */
}
```

Routine end( ) detects completion of byte write or block erase operations of a 28F008SA. It passes back the Status Register data it reads from the device. It also returns "0" if Status Register bit 7 indicates WSM "Ready", and "1" if indication is that the WSM is still "Busy".

```
/*****************************************************************************************************/
/*      Function: Eraschk                                                                            */
/*      Description: Completes full Status Register check for block erase (proper command            */
/*                sequence, Vpp low detect, block erase success). This routine assumes that block    */
/*                erase completion has already been checked in function end(), and therefore does     */
/*                not check the WSM Status bit of the Status Register                                 */
/*      Inputs:   statdata: Status Register data read in function end                                 */
/*      Outputs:  None                                                                                */
/*      Returns:  0 = Block erase completed successfully                                              */
/*                1 = Error; Vpp low detect                                                           */
/*                2 = Error; Block erase error                                                        */
/*                3 = Error; Improper command sequencing                                              */
/*      Device Read Mode on Return: Same as when entered                                              */
/*****************************************************************************************************/

#define   ESEQMASK  0X30              /* Mask to isolate the erase and byte write status bits of   */
                                      /*       the Status Register                                 */
#define   ESEQFAIL  0X30              /* Status Register value after masking if block erase        */
                                      /*       command sequence error has been detected            */
#define   EERRMSK   0X20              /* Mask to isolate the erase status bit of the               */
                                      /*       Status Register                                     */
#define   ERASERR   0X20              /* Status Register value after masking if block erase error  */
                                      /*       has been detected                                   */
#define   VLOWMASK  0X08              /* Mask to isolate the Vpp status bit of the Status Register */
#define   VPPLOW    0X08              /* Status Register value after masking if Vpp low            */
                                      /*       has been detected                                   */

int eraschk(statdata)

byte statdata;                        /* Status Register data that has been already read from the  */
                                      /*       28F008SA in function end()                          */

{
        if ((statdata & VLOWMASK) == VPPLOW)
                return (1);           /* Vpp low detect error, return code "1"                     */
        if ((statdata & EERRMSK) == ERASERR)
                return (2);           /* Block erase error detect, return code "2"                 */
        if ((statdata & ESEQMASK) == ESEQFAIL)
                return (3);           /* Block erase command sequence error, return code "3"       */
        return (0);                   /* Block erase success, return code "0"                      */
}
```

Routine eraschk( ) takes the Status Register data read in end( ) and further analyzes it. It returns "0" if block erase was successful, "1" if V_{PP} low error was detected, "2" if block erase error was reported and "3" if an erase command sequence error was found (erase setup followed by a command other than erase confirm). This is useful after a block or string of blocks has been erased, to check for successful completion.

```
/*****************************************************************************************************/
/*      Function: Writebgn                                                                           */
/*      Description: Begins byte write sequence                                                      */
/*      Inputs:   wdata: Data to be written into the device                                          */
/*                waddr: Target address to be written                                                */
/*      Outputs:  None                                                                               */
/*      Returns:  0 = Byte write successfully initiated                                              */
/*                1 = Byte write not initiated (ID check error)                                      */
/*      Device Read Mode on Return: Status Register (ID if returns 1)                                */
/*****************************************************************************************************/

#define   SETUPCMD  0X40              /* Byte Write Setup command                                   */

int writebgn(wdata,waddr)

byte wdata;                           /* Data to be written into the 28F008SA                       */
byte far *waddr;                      /* waddr is the destination address for the data              */
                                      /*       to be written                                        */

{
        byte mfgrid,deviceid;

        if (idread(&mfgrid,&deviceid)==1)  /* Device ID read error...powered up?                    */
                return (1);
        *waddr    = SETUPCMD;         /* Write Byte Write Setup command and destination address     */
        *waddr    = wdata;            /* Write byte write data and destination address              */
        return (0);
}

/*****************************************************************************************************/
/*      Function: Writechk                                                                           */
/*      Description: Completes full Status Register check for byte write (Vpp low detect, byte       */
/*                write success). This routine assumes that byte write completion has already        */
/*                been checked in function end() and therefore does not check the WSM Status         */
/*                bit of the Status Register                                                         */
/*      Inputs:   statdata: Status Register data read in function end()                              */
/*      Outputs:  None                                                                               */
/*      Returns:  0 = Byte write completed successfully                                              */
/*                1 = Error; Vpp low detect                                                          */
/*                2 = Error; Byte write error                                                        */
/*      Device Read Mode on Return: Status Register                                                  */
/*****************************************************************************************************/

#define   WERRMSK   0X10              /* Mask to isolate the byte write error bit of the            */
                                      /*       Status Register                                      */
#define   WRITERR   0X10              /* Status Register value after masking if byte write error    */
                                      /*       has been detected                                    */
#define   VLOWMASK  0X08              /* Mask to isolate the Vpp status bit of the                  */
                                      /*       Status Register                                      */
#define   VPPLOW    0X08              /* Status Register value after masking if Vpp low             */
                                      /*       has been detected                                    */

int writechk(statdata)

byte statdata;                        /* Status Register data that has been already read from the   */
                                      /*       28F008SA in function end()                           */
{
        if ((statdata & VLOWMASK) == VPPLOW)
                return (1);           /* Vpp low detect error, return code "1"                      */
        if ((statdata & WERRMSK) == WRITERR)
                return (2);           /* Byte write error detect, return code "2"                   */
        return (0);                   /* Byte/string write success, return code "0"                 */
}
```

Routine writebgn( ) issues a byte write command sequence to a 28F008SA. It is passed the desired system address for the byte to be written, as well as the data to be written there. After calling idread( ), it writes the byte write command sequence at the specified address. It returns "0" if byte write initiation was successful, and "1" if the ID read fails (device not powered up or $\overline{PWD}$ not disabled).

Routine writechk( ) takes the Status Register data read in end( ) and further analyzes it. It returns "0" if byte write was successful, "1" if V_{PP} low error was detected, and "2" if byte write error was reported. This is useful after a byte or string of bytes has been written, to check for successful completion.

```
/*****************************************************************************************************/
/*        Function: Idread                                                                           */
/*        Description: Reads the manufacturer and device IDs from the target 28F008SA                */
/*        Inputs:   None                                                                             */
/*        Outputs:  mfgrid: Returned manufacturer ID                                                 */
/*                  deviceid: Returned device ID                                                     */
/*        Returns:  0 = ID read correct                                                              */
/*                  1 = Wrong or no ID                                                               */
/*        Device Read Mode on Return: inteligent Identifier                                          */
/*****************************************************************************************************/

#define   MFGRADDR  0                   /* Address "0" for the target 28F008SA...alterable depending */
                                        /*        on the system architecture                         */
#define   DEVICADD  1                   /* Address "1" for the target 28F008SA...alterable depending */
                                        /*        on the system architecture                         */
#define   IDRDCOMM  0X90                /* Inteligent Identifier command                             */
#define   INTELID   0X89                /* Manufacturer ID for Intel devices                         */
#define   DVCID     0X0A2               /* Device IDs for 28F008SA                                   */
#define   DVCID2    0X0A1

int idread(mfgrid,deviceid)

byte *mfgrid;                           /* The manufacturer ID read by this function, to be          */
                                        /*        transferred back to the calling program            */
byte *deviceid;                         /* The device ID read by this function, to be transferred    */
                                        /*        back to the calling function                       */
{
          byte far *tempaddr;           /* Pointer address variable used to read IDs                 */
          tempaddr  = (byte far *)MFGRADDR;
          *tempaddr = IDRDCOMM;         /* Write inteligent Identifier command to an address within  */
                                        /*        the 28F008SA memory map (in this case 00 hex)      */
          *mfgrid   = *tempaddr;        /* Read mfgr ID, tempaddr still points at address "0"        */
          tempaddr  = (byte far *)DEVICADD;  /* Point to address "1" for the device specific ID      */
          *deviceid = *tempaddr;        /* Read device ID                                            */
          if ((*mfgrid != INTELID) || ((*deviceid != DVCID) && (*deviceid != DVCID2)))
                    return (1);         /* ID read error; device powered up?                         */
          return (0);
}
```

Routine idread( ) issues the Intelligent Identifier command to a 28F008SA. It passes back the manufacturer and device IDs it reads. In addition, it returns "0" if the IDs read matched those expected for the 28F008SA or 28F008SA-L, and "1" if either the manufacturer or device IDs did not match.

```
/*****************************************************************************************/
/*      Function: Statrd                                                                 */
/*      Description: Returns contents of the target 28F008SA Status Register            */
/*      Inputs:   None                                                                   */
/*      Outputs:  statdata: Returned Status Register data                                */
/*      Returns:  Nothing                                                                */
/*      Device Read Mode on Return: Status Register                                      */
/*****************************************************************************************/
#define   STATREAD  0X70              /* Read Status Register command                    */
#define   SYSADDR   0                 /* This constant can be initialized to any address within  */
                                      /*       the memory map of the target 28F008SA and is      */
                                      /*       alterable depending on the system architecture    */

int statrd(statdata)

byte *statdata;                       /* Allows Status Register data to be passed back to the    */
                                      /*       calling program for further analysis              */
{
        byte far *stataddr;           /* Pointer variable used to write commands to device       */

        stataddr  = (byte far *)SYSADDR;
        *stataddr = STATREAD;         /* Write Read Status Register command to 28F008SA          */
        *statdata = *stataddr;
        return;
}

/*****************************************************************************************/
/*      Function: Statclr                                                                */
/*      Description: Clears the 28F008SA Status Register                                 */
/*      Inputs:   None                                                                   */
/*      Outputs:  None                                                                   */
/*      Returns:  Nothing                                                                */
/*      Device Read Mode on Return: Array                                                */
/*****************************************************************************************/

#define   STATCLER  0X50              /* Clear Status Register command                   */
#define   SYSADDR   0                 /* This constant can be initialized to any address within  */
                                      /*       the memory map of the target 28F008SA and is      */
                                      /*       alterable depending on the system architecture    */

int statclr()

{
        byte far *stataddr;           /* Pointer variable used to write commands to device       */

        stataddr  = (byte far *)SYSADDR;
        *stataddr = STATCLER;         /* Write Clear Status Register command to 28F008SA         */
        return;
}
```

Routine statrd( ) reads a 28F008SA Status Register. It issues the Read Status Register command and passes back the data it obtains.

Routine statclr( ) issues the Clear Status Register command to a 28F008SA. This routine is required after analyzing Status Register contents in routines like eraschk( ) and writechk( ). The 28F008SA Status Register retains state of bits 3–6 until they are cleared by the Clear Status Register command.

```
/*****************************************************************************************************/
/*        Function: Rdmode                                                                           */
/*        Description: Puts the target 28F008SA in Read Array Mode. This function might be used, for */
/*                  example, to prepare the system for return to code execution out of the Flash     */
/*                  memory after byte write or block erase algorithms have been executed off-chip    */
/*        Inputs:   None                                                                             */
/*        Outputs:  None                                                                             */
/*        Returns:  Nothing                                                                          */
/*        Device Read Mode on Return: Array                                                          */
/*****************************************************************************************************/

#define   RDARRAY   0XFF                /* Read Array command                                       */
#define   SYSADDR   0                   /* This constant can be initialized to any address within   */
                                        /*        the memory map of the target 28F008SA and is      */
                                        /*        alterable depending on the system architecture    */

int rdmode()

{
        byte far *tempaddr;             /* Pointer variable used to write commands to the device    */

        tempaddr  = (byte far *)SYSADDR;
        *tempaddr = RDARRAY;            /* Write Read Array command to 28F008SA                     */
        return;
}

/*****************************************************************************************************/
/*        Function: Rdbyte                                                                           */
/*        Description: Reads a byte of data from a specified address and returns it to the           */
/*                     calling program                                                               */
/*        Inputs:   raddr: Target address to be read from                                            */
/*        Outputs:  rdata: Data at the specified address                                             */
/*        Returns:  Nothing                                                                          */
/*        Device Read Mode on Return: Array                                                          */
/*****************************************************************************************************/

#define   RDARRAY   0XFF                /* Read array command                                       */

int rdbyte(rdata,raddr)
byte *rdata;                            /* Returns data read from the device at specified address   */
byte far *raddr;                        /* Raddr is the target address to be read from              */

{
        *raddr    = RDARRAY;            /* Write read array command to an address within the        */
                                        /*        28F008SA (in this case the target address)        */
        *rdata    = *raddr;             /* Read from the specified address and store                */
        return;
}
```

Routine rdmode( ) simply puts a 28F008SA in Read Array mode. This is useful after byte write and block erase operations, to return the 28F008SA to its "normal" mode of operation. After block erase or byte write, the 28F008SA will continue to output Status Register data until the Read Array command is issued to it, for example.

Routine rdbyte( ) not only puts the 28F008SA in Read Array mode, it also reads a byte of data. It is passed the desired system byte address, and passes back the data at that address.

```
/*****************************************************************************************/
/*      Function: Vppup                                                                  */
/*      Description: Ramps the Vpp supply to the target 28F008SA to enable byte write or block  */
/*              erase. This routine can be tailored to the individual system architecture. For  */
/*              purposes of this example, I assumed that a system Control Register existed at  */
/*              system address 20000 hex, with the following definitions:                */
/*                                                                                       */
/*                      Bit 7: Vpph Control: 1 = Enabled                                 */
/*                                           0 = Disabled                                */
/*                      Bit 6: PWD Control:  1 = PowerDown Enabled                       */
/*                                           0 = PowerDown Disabled                      */
/*                      Bits 5-0: Undefined                                              */
/*                                                                                       */
/*      Inputs:   None                                                                   */
/*      Outputs:  None                                                                   */
/*      Returns:  Nothing                                                                */
/*      Device Read Mode on Return: As existed before entering the function. Part is now ready for */
/*              program or erase                                                         */
/*****************************************************************************************/

#define   VPPHIGH   0X80              /* Bit 7 = 1, Vpp elevated to Vpph                 */
#define   SYSCADDR  0X20000           /* Assumed system Control Register Address         */

int vppup()

{
        byte far *contaddr;           /* Pointer variable used to write data to the system  */
                                      /*        Control Register                         */

        contaddr  = (byte far *)SYSCADDR;
        *contaddr = *contaddr | VPPHIGH;  /* Read current Control Register data, "OR" with  */
                                      /*        constant to ramp Vpp                     */
        return;
}

/*****************************************************************************************/
/*      Function: Vppdown                                                                */
/*      Description: Ramps down the Vpp supply to the target 28F008SA to disable byte write/block */
/*              erase. See above for a description of the assumed system Control Register. */
/*      Inputs:   None                                                                   */
/*      Outputs:  None                                                                   */
/*      Returns:  Nothing                                                                */
/*      Device Read Mode on Return: As existed before entering the function. Part now has high Vpp */
/*              disabled. If byte write or block erase was in progress when this function was  */
/*              called, it will complete unsuccessfully with Vpp low error in the        */
/*              Status Register.                                                         */
/*****************************************************************************************/

#define   VPPDWN    0X7F              /* Bit 7 = 0, Vpp lowered to Vppl                  */
#define   SYSCADDR  0X20000           /* Assumed system Control Register Address         */

int vppdown()

{
        byte far *contaddr;           /* Pointer variable used to write data to the system  */
                                      /*        Control Register                         */

        contaddr  = (byte far *)SYSCADDR;
        *contaddr = *contaddr & VPPDWN;  /* Read current Control Register data, "AND" with  */
                                      /*        constant to lower Vpp                    */
        return;
}
```

Functions vppup( ) and vppdown( ) give examples of how to control via software the hardware that enables or disables 12V V_{PP} to a 28F008SA. The actual hardware implementation chosen will drive any modification of these routines.

```
/*****************************************************************************************************/
/*      Function: Pwden                                                                              */
/*      Description: Toggles the 28F008SA /PWD pin low to put the device in Deep PowerDown mode.     */
/*                See above for a description of the assumed system Control Register.                */
/*      Inputs:   None                                                                               */
/*      Outputs:  None                                                                               */
/*      Returns:  Nothing                                                                            */
/*      Device Read Mode on Return: The part is powered down. If byte write or block erase was in    */
/*                progress when this function was called, it will abort with resulting partially     */
/*                written or erased data. Recovery in the form of repeat of byte write or block      */
/*                erase will be required once the part transitions out of powerdown, to              */
/*                initialize data to a known state.                                                  */
/*****************************************************************************************************/

#define   PWD       0X40              /* Bit 6 = 1, /PWD enabled                                    */
#define   SYSCADDR  0X20000           /* Assumed system Control Register Address                    */

int pwden()

{
        byte far *contaddr;           /* Pointer variable used to write data to the system
                                      /*        Control Register                                    */

        contaddr  = (byte far *)SYSCADDR;
        *contaddr = *contaddr | PWD;  /* Read current Control Register data, "OR" with constant     */
                                      /*        to enable Deep PowerDown                            */
        return;
}

/*****************************************************************************************************/
/*      Function: Pwddis                                                                             */
/*      Description: Toggles the 28F008SA /PWD pin high to transition the part out of Deep           */
/*                PowerDown. See above for a description of the assumed system Control Register.     */
/*      Inputs:   None                                                                               */
/*      Outputs:  None                                                                               */
/*      Returns:  Nothing                                                                            */
/*      Device Read Mode on Return: Read Array mode. Low voltage is removed from the /PWD pin.       */
/*                28F008SA output pins will output valid data time tPHQV after the /PWD pin          */
/*                transitions high (reference the datasheet AC Read Characteristics) assuming        */
/*                valid states on all other control and power supply pins.                           */
/*****************************************************************************************************/

#define   PWDOFF    0XBF              /* Bit 6 = 0, /PWD disabled                                   */
#define   SYSCADDR  0X20000           /* Assumed system Control Register Address                    */

int pwddis()

{
        byte far *contaddr;           /* Pointer variable used to write data to the system          */
                                      /*        Control Register                                    */

        contaddr  = (byte far *)SYSCADDR;
        *contaddr = *contaddr & PWDOFF;  /* Read current Control Register data, "AND" with          */
                                         /*        constant to disable Deep PowerDown              */
        return;
}
```

Functions pwden( ) and pwddis( ) give examples of how to control via software the hardware that enables or disables a 28F008SA $\overline{\text{PWD}}$ input. The actual hardware implementation chosen will drive any modification of these routines.

ADDITIONAL INFORMATION

		Order Number
	28F008SA Datasheet	290429
	28F008SA-L Datasheet	290435
AP-359	"28F008SA Hardware Interfacing"	292094
AP-364	"28F008SA Automation and Algorithms"	292099
ER-27	"The Intel 28F008SA Flash Memory"	294011
ER-28	"ETOX™-III Flash Memory Technology"	294012

REVISION HISTORY

Number	Description
002	Revised Erase Suspend Algorithm in "C" Drivers.